リファレンス，配列，例外処理 その他演習に役立つこと

2004年7月21日

海谷 治彦

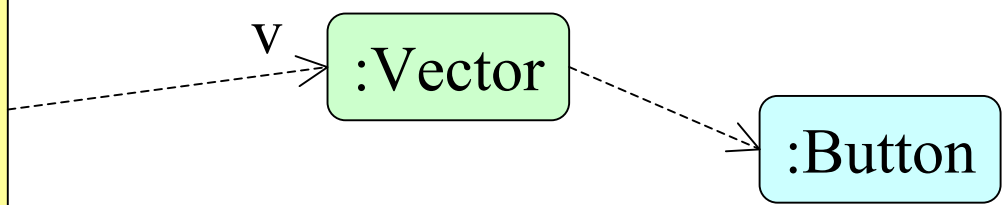
リファレンス

- ま，改め紹介しなくてもJava遣いなら誰でもつかってる.
- インスタンスをプログラム中から識別(捕獲)するためのラベルのようなもの.
- Cでいうところのポインタ変数に相当.
- Javaでは，あるインスタンスを参照するリファレンスが1つもなくなると，勝手にインスタンスは消去される.
  - AutoGC(自動ゴミ集め)機能.

例

```
・・・・
Vector v;
   v=new Vector(); // ココでインスタンスができる
・・・・・
   v.add(new Button());
・・・・・・
```

別のインスタンス
もしくは
staticメソッド



プログラム内では，インスタンス固有の名前ではなく，
あるインスタンスから見た相対的な名前で処理をする．
そのような相対的な名前が「リファレンス」

複数リファレンスで1つのインスタンスを指す

```
public class UnivClass {

private Student student;

public static void main(String[] args) {
UnivClass uc=new UnivClass();
Student s=new Student();
s.setUnivClass(uc);
}
public Student getStudent() {
return student;
}

public void setStudent(Student student) {
this.student = student;
}

}
```

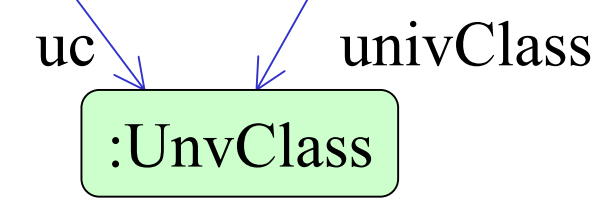
```
public class Student {

private UnivClass univClass;

public static void main(String[] args) {
}
public UnivClass getUnivClass() {
return univClass;
}

public void setUnivClass(UnivClass univClass) {
this.univClass = univClass;
}

}
```

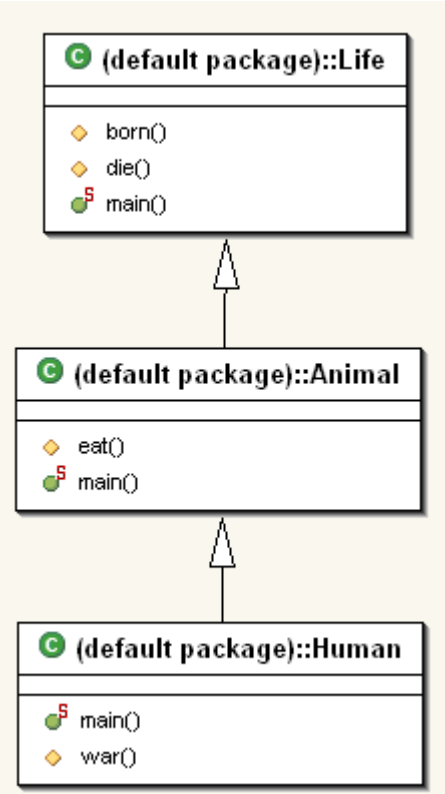


異なるクラス間で同一インスタンスの共有ができる.

リファレンスの型付け

- リファレンスはどのクラスを指すかの型情報を持つ.
- 基本的にあるクラス(型)のインスタンスは, 同じクラス(型)のリファレンスで指す.
- しかし, スーパークラス(もしくはインタフェース)のリファレンスで指してもよい.
  - 無論, 使えるメソッドは少なくなるが.

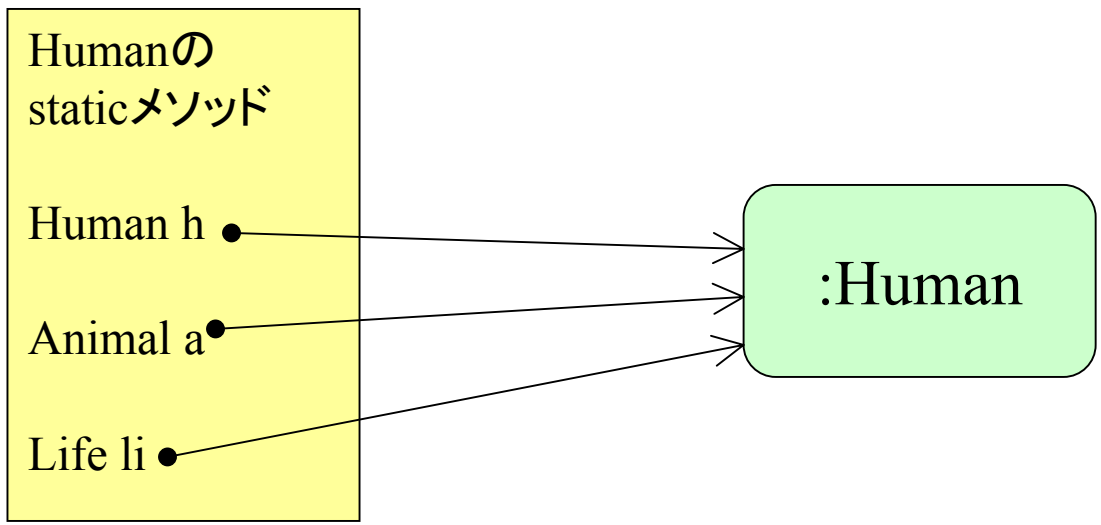
例



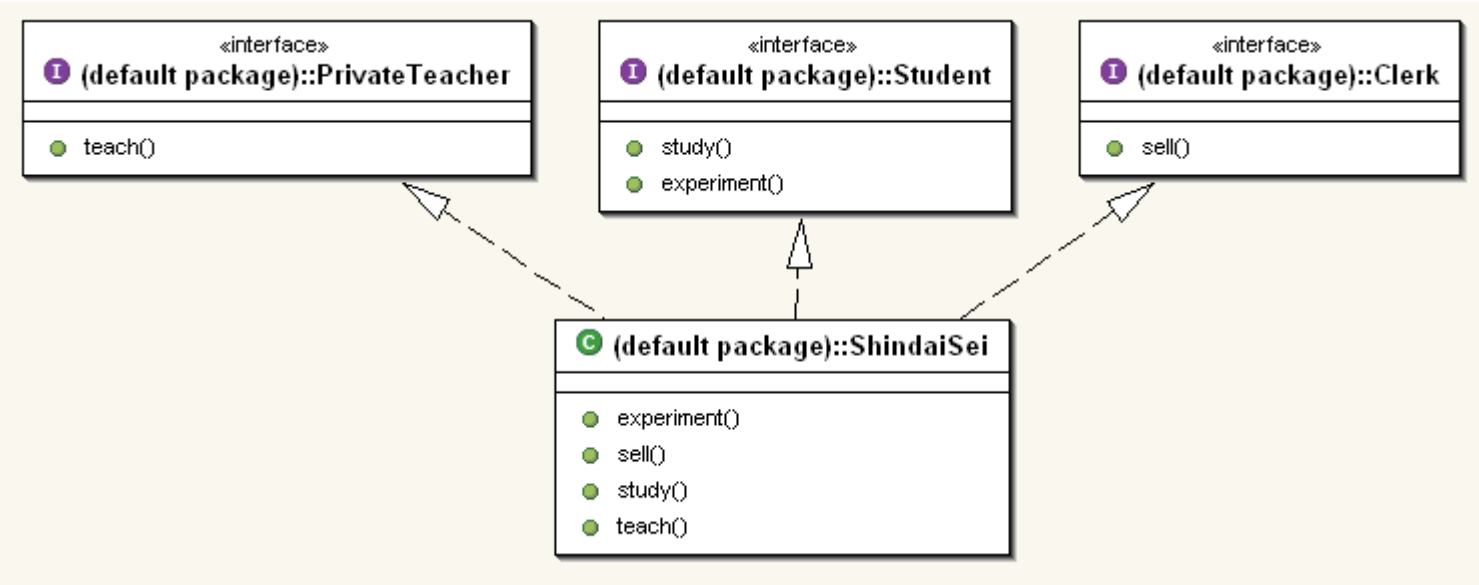

```
public class Human extends Animal {

  public static void main(String[] args) {
   Human h=new Human();
    Animal a=h;
    Life li=h;
  }
  protected void war() {}

}
```



インタフェースでも同様




```
public class ShindaiSei implements Student, PrivateTeacher, Clerk {

public static void main(String[] args) {
ShindaiSei ss=new ShindaiSei();
Student s=ss;
PrivateTeacher pt=ss;
Clerk c=ss;
}
```

配列について

- 別途URLを参照.

例外処理 Exception

- 名前の通り通常ではない例外的，というか異常事態を処理するための機構.
- 現代的なプログラミング言語はほとんどこの機構を持つ.
- メソッド(関数)の処理が最後まで実行されず途中でコケるような事態の処理を想定している.
  - ゼロで割り算.
  - 配列の添え字の範囲を超える.
  - (入出力)装置の異常.

  等

例外を処理しないと？

- プログラムが途中で異常終了する.
- C言語等では異常終了するような要因を抑え込んで, プログラムを続行する手段がなかった.
  - signal等, エグい手を使えばできないことはないが.

例外を捕まえる

- API内クラスの一部メソッドは特定の状況下で例外を発生(投げる)ようになっている.
- 投げられた例外を適切に捕まえる(catch)するスキルがまず必要.
- 基本的には以下のような文法.

```
try{
    例外発生の可能性があるメソッド呼び出しの列
}catch(ある例外型 exp){
    発生した例外情報をもつインスタンスへの
    リファレンスexpを使い,
    例外事態を処理する. (通常は無視か・・・)
}
```

例外を投げるメソッドの例

java.io
クラス BufferedReader
java.lang.Object
  |
  +--java.io.Reader
        |
        +--java.io.BufferedReader

readLine
public String **readLine**() throws IOException
　　1 行のテキストを読み込みます。1 行の終端は、改行 ('¥n') か、復帰 ('¥r')、または復行とそれに続く改行のどれかで認識されます。
　　戻り値:
　　行の内容を含む文字列、ただし行の終端文字は含めない。ストリームの終わりに達している場合は null
例外:
　　IOException - 入出力エラーが発生した場合

例: stdinから文字を読む

```
import java.io.*;
// 中略
InputStreamReader  isr=new InputStreamReader(System.in);
   BufferedReader br=new BufferedReader(isr);
   int c=0;
   while(true){
     String s=null;
     try{
       s=br.readLine();    // ココが例外IOExceptionを投げる可能性のある部分
     }catch(IOException e){
       e.printStackTrace();   // 結果として例外の経緯を表示して,
       break;                 // ループを抜ける対処をしている.
     }
     System.out.println(c+": "+s);
     c++;
     if(s.length()==0) break;
   }
```

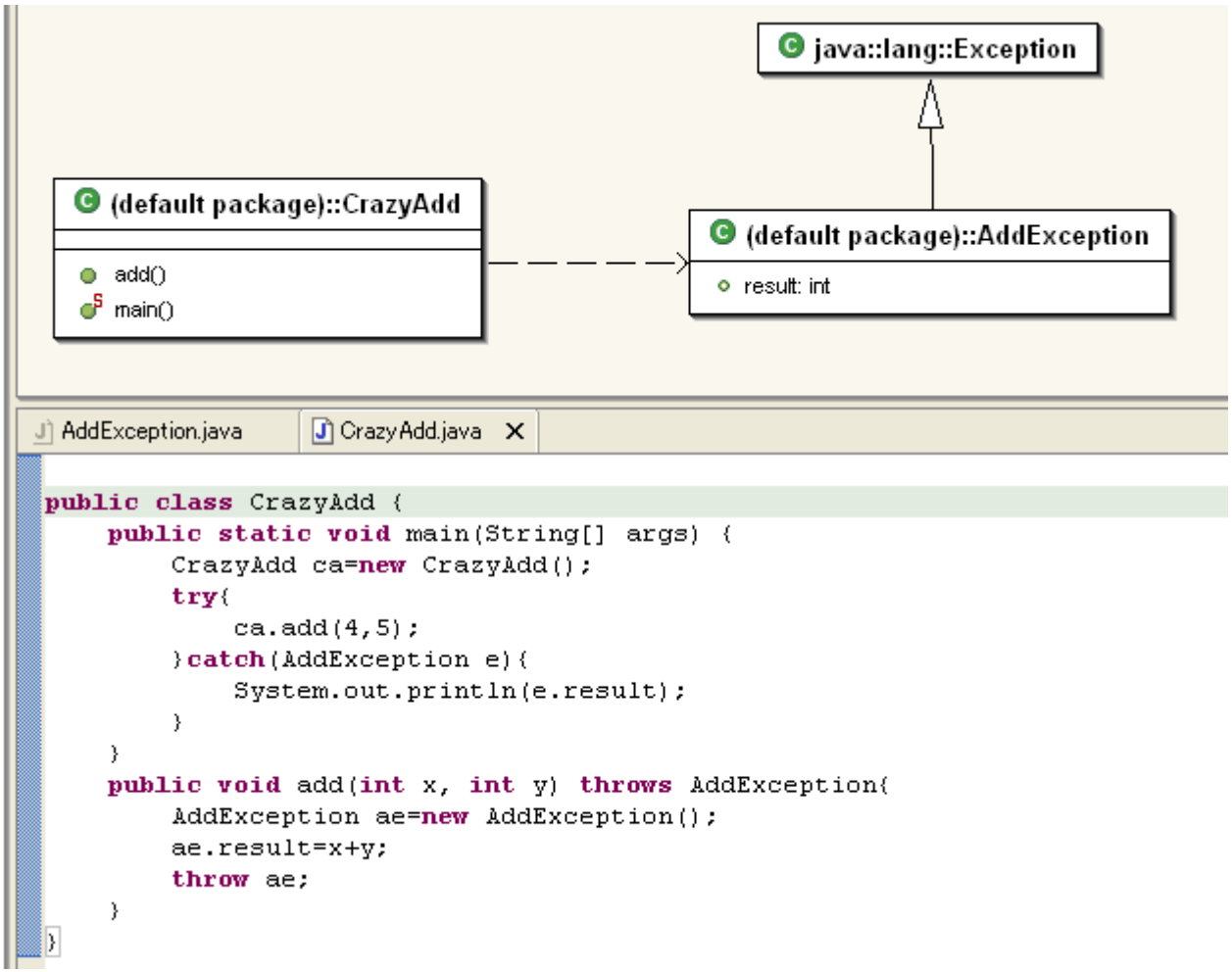
ここでの表現は文字列を取得する際の定型です.

例外処理と正常処理との違い

- 例外と正常は「意味領域」の話なので，ある計算結果がどちらになるかは主観的.
  - というか「仕様」で何が例外で何が正常かが決まる.
- よって結果を返り値とするか例外とするかの文法的なガイドラインは無い.
  - 例外インスタンスとして計算結果を返すようなプログラミングも可能である. (無論，変なプログラム)
  - C言語では例外(例えばファイルが開けられない等)もすべて返り値で結果通知をしていた.
- しかし,
  - なんとなし正常っぽいものは返り値.
  - なんとなし異常っぽいものは例外.

  でメソッド呼び出し側に返すのがよい.

悪い例: 計算結果を例外で返す




```
public class CrazyAdd {
    public static void main(String[] args) {
        CrazyAdd ca=new CrazyAdd();
        try{
            ca.add(4,5);
        }catch(AddException e){
            System.out.println(e.result);
        }
    }
    public void add(int x, int y) throws AddException{
        AddException ae=new AddException();
        ae.result=x+y;
        throw ae;
    }
}
```

例外クラスを自分で作る

- 前項の例がそうだが，当面初心者は手を出す必要がないと思う.
  - 文法は単純だが意味の理解が難しい.
- 当面はAPI内のクラスにあるメソッドが投げる例外を正しく捕獲(catch)できるスキルを磨こう.

Collectionの実装

- Collectionインタフェースを実装したクラス群のこと.
  - API内にもかなり沢山ある.
- いわゆる「可変長配列」や「集合」を扱うことができるので，とっても便利.
  - Cでせこせこリスト構造を作っていたのがバカらしくなる.
- 代表例.
  - HashMap, ArrayList, Vector, HashSet
  - Vectorの利用は昨今，お勧めでないらしい.

例 HashSet クラス

java.util
クラス HashSet
java.lang.Object
  |
  +--java.util.AbstractCollection
        |
        +--java.util.AbstractSet
              |
              +--java.util.HashSet
すべての実装インタフェース:
Cloneable, Collection, Serializable, Set
直系の既知のサブクラス: JobStateReasons, LinkedHashSet

要は数学における「集合」に相当するものを表現できるクラス.

HashSetの主なメソッド

- boolean add(Object o) oをメンバーに入れる
  $S \Rightarrow S \cup \{o\}$
- boolean remove(Object o) 削除
  $S \Rightarrow S \setminus \{o\}$
- boolean contains(Object o) 含むか否かを判定
  $S \ni o$
- boolean containsAll(Collection c) 包含関係
  $S \supset c$
- int size() 要素数. いわゆるcardinarity
  $|S|$
- Iterator iterator() 反復子を返す. (後述)

Iterator 反復子

- 昨今，添え字を使い集まり(集合や配列等)を列挙するより，Iterator というので列挙するほうが推奨されいる.
- 添え字での列挙が推奨されない理由？
  - 配列等のように添え字があり順序付けされたもの以外の集まりを扱えない.
    - 例えば，リンクドリスト(リスト構造)とか集合とか.

Interface Iterator

- 以下のメソッドの実装を指示している.
  - boolean hasNext()
    - 繰り返し処理で次の要素が存在すればtrue, なければfalse.
  - Object next()
    - 次の要素を返す.

  他
- Collectionインタフェースを実装したクラスは，メソッド Iterator iterator() を持ち，反復子による列挙が可能.

使い方の例

```
import java.util.*;

public class aa {
    public static void main(String[] args){
        HashSet hs=new HashSet();
        hs.add("sakana");
        hs.add("tako");
        hs.add("unagi");

        // for を使う繰り返し
        for(Iterator i=hs.iterator(); i.hasNext(); ){
            System.out.println(i.next());
        }

        // whileを使う繰り返し
        Iterator i=hs.iterator();
        while(i.hasNext()){
            System.out.println(i.next());
        }
    }
}
```

Console [<terminated> C:¥Program Files¥Java¥j2re1.4.2_03¥bin¥javaw.exe (04/07/21 3:5

```
sakana
tako
unagi
sakana
tako
unagi
```